Spi2Java User Manual

Alfredo Pironti, Davide Pozza, Riccardo Sisto

Politecnico di Torino
C.so Duca degli Abruzzi, 24
10129 - Torino - Italy

February 1, 2008

Disclaimer: Please, remember that the Spi2Java framework is a research prototype. We are already conscious of the presence of some bugs and security problems. We will fix them as soon as possible. Please feel free to contact us by email if you encounter or find any problem. Feedbacks and suggestions are also very appreciated. When you contact us, please send the email to all of us

`{davide.pozza,alfredo.pironti,riccardo.sisto}@polito.it`

and use "Spi2Java" as a prefix in the subject. We will answer as soon as possible.

1 Introduction

Formal methods have the potential to significantly improve the trustworthiness of critical software artifacts, especially when development turns out to be intrinsically error prone, and bugs difficult to discover, as it happens with cryptographic protocols. However, one of the problems that still limits the widespread use of formal methods is the high level of expertise that they normally require and the high cost of development that their use implies. A way to partially overcome the above problem and to improve the acceptance and productivity of formal methods is to provide methodologies and tools that simplify the use of formal methods, introducing automation and hiding underlying complexity.

Spi2Java is a set of tools that support a model-based approach to cryptographic protocol implementation which can take advantage of the capabilities of high-level formal analysis tools [1] and also gives support in order to ensure the correspondence between high level formal models and their implementations. In particular, Spi2Java is a set of tools to produce Java implementations of cryptographic protocols described in the spi calculus formal language.

In short, here we introduce the main Spi2Java tools and their aims.

spi preprocessor is a pre-processor and parser for the spi calclulus description of the protocol.

spi2java refiner is a type checker that infers Java types for the spi calculus terms used in the protocol description.

Table 1: Term syntax of Spi-Calculus

σ, ρ	terms
m	name
(σ, ρ)	pair
0	zero
$\mathrm{suc}(\sigma)$	successor
x	variable
$\mathrm{H}(\sigma)$	hashing
$\{\sigma\}_\rho$	shared-key encryption
$\sigma^+,\ \sigma^-$	public/private part
$\{[\sigma]\}_\rho$	public-key encryption
$[\{\sigma\}]_\rho$	private-key signature

spi2java code generator is an automatic code generator that emits the Java code implementing the protocol described in spi caluclus.

The rest of this document is organized as it follows. Section 2 gives an informal introduction to the spi calculus language and presents a reference example that will be used throughout this document. Section 3 shows in detail, through the full development of the reference example, how the tool can be used to produce an implementation of a cryptographic protocol. Section 4 explains how to use some of the correlated tools.

2 Introducing the Spi Calculus and a Reference Example

The spi calulus is defined in [2] as an extension of the π calculus [3] with cryptographic primitives. It is a process algebraic language designed for describing and analyzing cryptographic protocols. These protocols heavily rely on cryptography and on message exchange through communication channels; accordingly, the Spi-Calculus provides powerful primitives to express cryptography and communication.

This section summarizes the syntax and describes the language semantics informally.

A spi calculus specification is a system of independent processes, executing in parallel; they synchronize via message-passing through named communication channels. The spi calculus has two basic language elements: terms, to represent data, and processes, to represent behaviors.

Terms can be either atomic elements, i.e. names, including the special name 0 representing the integer constant zero, and variables, or compound terms built using the term composition operators listed in Tab. 1. Names can represent communication channels, atomic keys and key pairs, nonces (also called *fresh names*) and any other unstructured data.

The informal meaning of the composition operators is as follows:

- $(\sigma,\ \rho)$ is the *pairing* of σ and ρ. It is a compound term whose components are σ and ρ. Pairs can always be freely split into their components.

$P, Q, R ::=$	processes
$\overline{\sigma}\langle\rho\rangle.P$	output
$\sigma(x).P$	input
$P \mid Q$	composition
$(\nu b)\ P$	restriction
$!P$	replication
0	nil
$[\sigma\ is\ \rho]\ P$	match
$let\ (x, y) = \sigma\ in\ P$	pair splitting
$case\ \sigma\ of\ 0 : P\ \mathrm{suc}(x) : Q$	integer case
$case\ \eta\ of\ \{x\}_\rho\ in\ P$	shared-key decryption
$case\ \eta\ of\ \{[x]\}_\rho\ in\ P$	decryption
$case\ \eta\ of\ [\{x\}]_\rho\ in\ P$	signature-check

Table 2: Process Syntax of Spi-Calculus

- $\mathrm{suc}(\sigma)$ is the *successor* of σ. This operator has been introduced mainly to represent successors over integers, but it can be used, more generally, as the abstract representation of an invertible function on terms.
- $\mathrm{H}(\sigma)$ is the *hashing* of σ. $\mathrm{H}(\sigma)$ represents a function of σ that cannot be inverted.
- Term $\{\sigma\}_\rho$ is the ciphertext obtained by encrypting σ under key ρ using a shared-key cryptosystem.
- σ^+ and σ^- represent respectively the public and private half of a key pair σ. σ^+ cannot be deduced from σ^- and vice versa.
- $\{[\sigma]\}_\rho$ is the result of the public-key encryption of σ with ρ.
- $[\{\sigma\}]_\rho$ is the result of the signature (private key encryption) of σ with the private key ρ.

Besides term specification, spi calculus also offers a set of operators to build behavior expressions which, in turn, represent processes.

Tab. 2 shows the operators available to build behavior expressions. Their informal meaning is:

- $\overline{\sigma}\langle\rho\rangle.P$ is an *output process*, ready to output term ρ on channel σ when a synchronization occurs. The behavior after the synchronization is described by behavior expression P.
- $\sigma(x).P$ is an *input process*, ready to perform an input from channel σ when a synchronization occurs. The behavior after a synchronization in which the received message is term η is described by behavior expression P with any occurrence of x replaced by η, which is denoted $P[\eta/x]$.
- $P \mid Q$ is a *parallel composition* where P and Q run in parallel. They may either synchronize between themselves or with the external environment separately. This operator is commutative and associative.

- $(\nu b)P$ is a *restriction* process which makes a fresh, private name b and then behaves as described by P.
- $!P$ is a *replication* where an unbounded number of instances of P run in parallel.
- $[\sigma\ is\ \rho]P$ is a *match* process which behaves as described by P if the terms σ and ρ are the same, and is stuck otherwise.
- 0 is the *nil* process: it is a stuck process.
- $let\ (x,\ y)\ =\ \sigma\ in\ P$ is a *pair splitting* process. If term σ is a pair $(\rho,\ \theta)$, this process behaves as $P[\rho/x, \theta/y]$, otherwise it is stuck.
- $case\ \sigma\ of\ 0 : P\ \mathrm{suc}(x) : Q$ is an *integer case* process. If σ is 0, it behaves as P; if σ is $\mathrm{suc}(\rho)$, it behaves as $Q[\rho/x]$. It is stuck otherwise.
- $case\ \eta\ of\ \{x\}_\rho\ in\ P$ is a *shared-key decryption* process. If η is a cyphertext taking the form $\{\sigma\}_\rho$, it behaves as $P[\sigma/x]$, otherwise it is stuck.
- $case\ \eta\ of\ \{[x]\}_\rho\ in\ P$ is a *decryption* process and behaves as $P[\theta/x]$ if η represents a message θ encrypted under a public key whose corresponding private key is ρ. Otherwise it is stuck.
- $case\ \eta\ of\ [\{x\}]_\rho\ in\ P$ is a *signature check* process and behaves as $P[\theta/x]$ if η represents a message θ signed under a private key whose corresponding public key is ρ. Otherwise it is stuck.

2.1 An Example

Fig. 1 shows the spi calculus specification of the *Andrew* key exchange protocol, as it is accepted by our tool. It is worth noting that, to have ASCII specification files, some different typographic conventions have been introduced with respect to the original spi calculus:

- The restriction symbol (ν) is substituted with the `@` character and the overlining of channels in output statements is simply omitted.
- Comments begin with a '`#`' and extend until the end of the line.
- In order to avoid typos, free variables must be explicitly declared in a preamble, by listing them in the following way:

  ```
  free: <freeVar1>,<freeVar2>, ... , <freeVarN>.
  ```

  Note the dot '`.`' at the end of the variables declaration.

The specification is composed of two files, `andrewPA.txt` and `andrewPB.txt`, containing one process description each, namely `pA` and `pB`, which respectively represent the client and server roles of the protocol.

The comments show the exchanged messages using the informal, intuitive representation often encountered in the literature, where $A \rightarrow B : \sigma$ means that A sends message σ to B. The first column shows the corresponding spi calculus specification for process pA, whereas the second column shows the corresponding behavior of process pB.

AndrewPA.txt	*AndrewPB.txt*	*Common comments*
free: M, KeyStore, cAB, A, B.	free: KeyStore, cAB.	# Free variables # declaration
pA(M, KeyStore):= (@Na) cAB<(A,Na)>.	pB(KeyStore):= cAB(xA,xNa).	# A->B: A,Na
cAB(xMSG). KeyStore<B>. KeyStore(kAB). case xMSG of {xNa,xk1AB}kAB in [xNa is Na]	KeyStore<xA>. KeyStore(kAB). (@k1AB) cAB<{(xNa,k1AB)}kAB>.	# B->A: {(Na,k1AB)}kAB
cAB<{Na}xk1AB>.	cAB(xMSGcypher). case xMSGcypher of {xnewNa}k1AB in [xnewNa is xNa] KeyStore<xA,k1AB>.	# A->B: {Na}k1AB
cAB(dummy). KeyStore<B,xk1AB>.	(@Nb) cAB<Nb>.	# B->A: Nb
cAB<{M}xk1AB>.0	cAB(Mcypher). case Mcypher of {x}k1AB in 0	# A->B: {M}k1AB

Figure 1: The Andrew Protocol spi calculus specification

The Andrew protocol assumes that each process has a local key store where symmetric keys are stored. Since the key store explicitly partakes in the protocol, it must be modeled in spi calculus. A simple modeling strategy is to represent the key store as a separate process that interacts with the corresponding protocol principal through a dedicated communication channel (the *key store channel*) not accessible by the intruder. The operations of getting and storing a key are modeled as inputs and outputs on the key store channel respectively. More precisely, a key is stored in the key store under an alias, which permits its unique identification. So, the operation of retrieving a stored key is represented by the statements `KeyStore<xA>.KeyStore(kAB)` where `KeyStore` denotes the interaction channel, `xA` is the alias and `kAB` is the variable where the key extracted from the key store is saved. The corresponding storing operation is described by the statement `KeyStore<xA,k1AB>` where `k1AB` is the key that must be stored under the alias `xA`.

The figure only reports the specification of the processes representing the protocol principals, whereas the process representing the behavior of the key store is omitted.

In a run of the Andrew protocol, five messages are exchanged through a public communication channel named `cAB`:

1. `A` sends to `B` a Pair composed of its identifier `A` and a new Nonce `Na`.

2. When `B` receives the message, it retrieves the key `kAB` shared with `A` from its local Key Store and builds a new fresh key `k1AB`. Then it sends a Pair encrypted with the shared key `kAB` to `A`. The Pair is composed of the Nonce extracted from the received message (stored locally in variable `xNa`) and the new key `k1AB`. When `A` receives the reply from `B`, it gets the key shared with `B` from the local Key Store and uses this key to decrypt the message.

Then, it splits the pair components, stores the second component (i.e. the new session key `k1AB`) in its local variable `xk1AB`, and verifies that the first component equals `Na` (i.e. the nonce it had previously sent to `B`).

3. `A` sends the Nonce `Na` encrypted with the received key to `B`. `B` receives the message, decrypts it with key `k1AB`, and verifies that the received Nonce (assigned to variable `xnewNa`) is the same it had previously received. Then it stores the key `k1AB` in its local key store under the alias contained in variable `xA`, overwriting the previous stored key.

4. `B` sends a new fresh Nonce `Nb` to `A`. Upon receiving this Nonce, `A` stores the key it had previously received in the local keystore under the alias `B`.

5. `A` sends a secret Message `M` ciphered with the agreed key to `B`. `B` receives the Message, decrypts it, and puts the result in its local variable `x`.

It can be noted how spi calculus enables the precise specification of all the operations performed by the protocol principals, including their interactions with their local key stores.

3 Spi2Java Development Methodology and Tool Support

In this section the suggested approach to the model-based spi2java development process is shown. The reference example will be carried out while explaining how the tools can be used. Moreover, in [4] the interested reader can find a case study, where the spi2java framework is used to automatically generate an SSH Transport Layer Protocol client implementation.

A data flow diagram of the suggested development approach is reported in figure 2.

3.1 Hints for the Formal Verification Step

Formal verification of protocols is not the goal of the spi2java tools, nor of their user manual. Nevertheless, some hints to use the S^3A [1] formal verification tool are provided.

In order to produce a protocol description that is accepted by S^3A, free variables declaration and comments must be removed from code: this operation can be performed by using the provided *sourceCleaner* parser (see section 4). Moreover the S^3A tool requires that the models of all the actors are reported into a single file (say, `andrew.txt`), where the additional *protocol instance* and *KeyStore* processes are added:

```
Inst(M):=( ( (@KeyStore) (pA(M, KeyStore) | pKS(KeyStore)) ) |
           ( (@KeyStore) (pB(KeyStore) | pKS(KeyStore)) ) )
```

The obtained `andrew.txt` file can be finally passed to the S^3A tool.

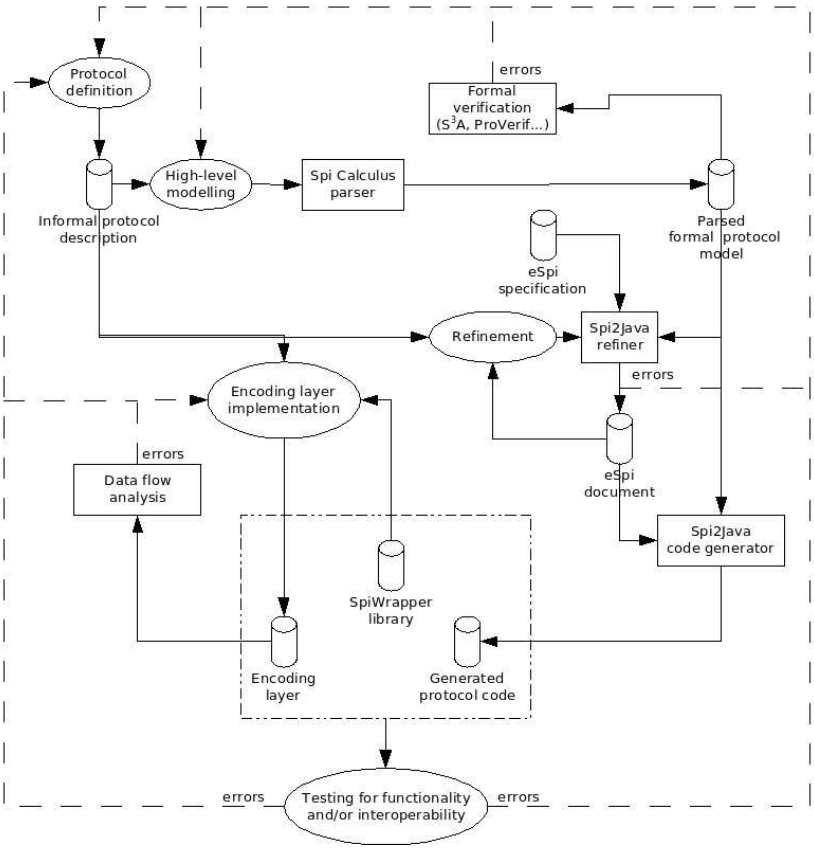

Figure 2: A data flow diagram of the suggested model-based development technique.

3.2 Writing the Formal Model

The programmer starts from an informal description of the protocol, and manually derives a formal spi calculus model. For the reference example, the informal protocol description, without low-level details, and the complete formal spi calculus model have already been described in the previous section.

The spi calculus model of each actor has been written into a separate plain text file: `andrewPA.txt` contains the model of actor A, and `andrewPB.txt` the model of actor B. By using the pre-processor and parser tool, called *spi0*, the syntax of the descriptions can be validated, and a symbol table for each actor is produced in output.

From now on, we assume that commands are invoked while being in the root directory of the spi2Java framework distribution. If this is not the case, you have to modify the paths according to your current working directory.

For example, we generate the symbol table for actors A and B by using the following commands

```
java -jar spi0.jar -i andrewPA.txt -o andrewPA.css
java -jar spi0.jar -i andrewPB.txt -o andrewPB.css
```

where `-i` indicates the input source file, and `-o` indicates the symbol table output file, which has extension `css`.

3.3 Refining the Formal Model

3.3.1 Theory

In order to derive a Java application from the spi calculus source, the spi2Java refiner can be used to fill the low-level implementation details that are abstracted away by the spi calculus language.

A tool like the spi2Java refiner can automatically infer *some* information about the missing details that are not present in the formal high-level model. For example, the type of certain data can be automatically inferred looking at how they are used. The implementation details that cannot be automatically inferred must be manually provided by the user, who can get them from the informal protocol description.

However, an interesting feature of the tool is the possibility to get early prototyping without any (or very few) manual intervention, just after having written the formal model. In order to get the early prototype, the tool can fill all the missing needed data with default values, which allows to immediately get a complete specification. The user can later change the default values to accommodate needs; after editing, the spi2Java refiner checks the user-given values for correctness and coherence with the reference spi calculus specification.

The low-level implementation details can be grouped into two main categories:

1. Cryptographic and Configuration parameters
2. Encoding/decoding functions (or, simply, encoding functions)

The first group of details specifies parameters such as “what algorithm must be used for a particular encryption operation” or “what network interface must be used by a particular channel”. In order to make the generated code compliant

with the implemented protocol, it is necessary that these parameters can be set independently, at compile time or at run time, for each data item.

The second group of details deals with the transformation from the internal representation of messages into their external representation, and vice versa. The internal representation is the one used to perform all the operations prescribed by the protocol logic on the data; whereas the external representation is the stream of bytes that must be exchanged with the other parties. Decoupling internal and external representations is necessary in order to obtain interoperability, because the external representation must conform to the agreed binary formats defined for the protocol.

Another task that the spi2Java refiner carries out is to statically assign a type to each spi calculus term. This is necessary because the spi calculus is an untyped language, while Java is statically typed. An extensible type hierarchy, reported in figure 3, has been defined for this purpose.

Here is a brief description of the meaning of types shown in figure 3.

Message is the most generic type: it represents an opaque message.

- **Name** is a partially specialized type that represents any atomic spi calculus term (i.e. a spi calculus name). Subtypes of this type are used to specialize the meaning of atomic data. Terms belonging to most of the subtypes of the Name type can be instantiated as fresh data.
  - **Channel** is the abstract representation of generic communication channels and has some extensions that are worth noting:
    - **Tcp/Ip Channel** provides access to the Tcp/Ip communication layers;
    - **Key Store Channel** provides access to an abstract keystore. One of its specialization is the *Java Key Store Channel* that uses the Java keystore implementation. Other implementations could be defined, such as an openSSH compatible keystore.
    - **File Channel** provides access to the local file system.
  - **Key Pair** represents a pair of public/private keys for use with asymmetric cryptosystems. Note that a key pair is still an atomic message, because public/private keys are obtained by $^+$ and $^-$ constructors, and not by pair splitting.
  - **Shared Key** represents a key for use with symmetric cryptosystems.
  - **Nonce** represents a randomly chosen sequence of bits.
  - **Identifier** represents some information which identifies an entity in a unique way. For example, it can be used as an alias to identify a key or a certificate stored inside a KeyStore. It cannot be instantiated fresh. In order to obtain a random string, the *Nonce* type should be used.
  - **Timestamp** represents a time snapshot.
- **Public Key** represents the public component of a key Pair.
  - **Certificate** represents a digital certificate. It is considered a specialization of Public Key because each digital certificate contains a public key, along with some other information.

Private Key represents the private component of a key Pair.

Hashing represents the result of applying a cryptographic hash function on some data (also known as message digest).

- **Cryptographic Hashing** represents the result of applying a cryptographic hash function, such as SHA-256 or MD5, to some data.
- **DH Hashing** represents the result of applying exponentiation g^x mod p

Shared Key Ciphered represents the result of a symmetric encryption performed using a Shared key:

Private Key Ciphered represents the result of an asymmetric encryption performed using a Private key.

Public Key Ciphered represents the result of an asymmetric encryption performed using a Public key.

Integer represents an integer number.

- **Integer With Bounds** represents an integer number that must fit within a given range.

Successor represents the logical successor of some data. (Currently only implemented as successor of an *Integer*.)

Pair represents a container of a pair of objects that can be of heterogeneous types. A tuple of objects is translated, inside the program, into nested Pair objects.

It must be pointed out that *Private Key* and *Public Key* are not atomic names because they are derived from a *Key Pair*.

The extensible type hierarchy allows new types to be added when the need arises. For instance, new channel extensions could be defined, in order to provide access to other communication layers, such as Udp/Ip, or to other system provided functions, such as databases.

Types are assigned by the spi2Java refiner using a set of *inference rules*, that, based upon the use of a term in the spi calculus model, assign it the correct type. As stated above, it is possible that the type of a term needs to be manually refined into a more specialized type. However, the spi2Java refiner checks that the type hierarchy is not infringed. For instance, if a term is automatically typed to *Channel*, it can be manually refined to the *Tcp/Ip Channel* or the the *Java Key Store Channel*, but it cannot be typed as *Message* or *Timestamp*.

Furthermore, there is a relationship between the type of a term and its associated low-level parameters. On one hand, each type has its own extensible set of cryptographic and configuration parameters. For instance, the *Shared Key* type has the *algorithm*, *strength* and *provider* parameters, which respectively specify the key cryptographic algorithm, the key strength and the Cryptographic provider that will implement the required cryptographic functions. On the other hand, for each type, an extensible set of Java classes can implement the encoding/decoding functions.

In order to store, for each spi calculus term, the assigned type and its low-level implementation details, the spi2Java refiner uses an eSpi (extended Spi) XML document, which is coupled with the original spi calculus source. The

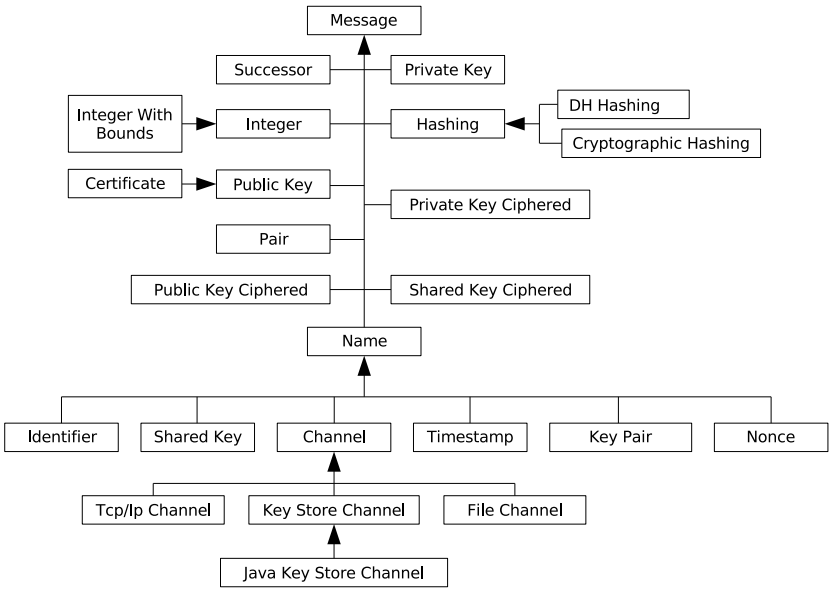

Figure 3: The currently defined eSpi type hierarchy.

eSpi document is automatically generated by the spi2Java refiner, and, when no information can be automatically inferred, default values are used.

In order to let the type hierarchy and the associated parameters be extensible, an XML document, called the eSpi specification, which contains these pieces of information, is parsed at run time by the spi2Java refiner; no information is hard coded into the tool. The allowed encoding layers and the default values of parameters are also stored into the eSpi specification, so that they can be modified at any moment too.

For further agile prototyping, a default encoding/decoding layer, which uses the Java serialization, is provided; however, in real environments, this default encoding/decoding layer has to be substituted with a user-given layer in order to implement the desired protocol encoding scheme.

Moreover, the spi2Java refiner allows the specification of cryptographic and configuration parameters to be done either statically at compile time, or dynamically at run time. The latter behavior enables the implementation of protocols that use the cryptographic algorithms in the way they have been negotiated at run-time by a cryptographic agreement pre-protocol handshake. In such case, the spi2Java refiner enables specifying that the value of a certain spi calculus term has to be used as a parameter for a cryptographic operation.

With respect to the data flow diagram in figure 2, the spi2Java refiner can be used in order to obtain the initial eSpi document coupled with the spi calculus protocol that is going to be implemented.

The steps that must be accomplished in order to obtain the client side and the server side eSpi documents are the same for each side. For this reason only the steps for the client side are reported here in detail. The server side will be referred only when it significantly differs from the client side.

In particular, the eSpi document for the *client* process can be obtained in two steps.

Step 1 Only the spi calculus model and the eSpi specification are given as input to the spi2Java refiner. The tool generates an eSpi document using all the information that can be automatically inferred from the given model. In particular, types are assigned to terms based on their use, while cryptographic and configuration parameters and the encoding layer are set to their defaults.

Step 2
1. The generated eSpi document is manually refined, adding the needed information that could not be automatically inferred.
2. The spi2Java refiner runs again, taking the manually modified eSpi document, the spi calculus model and the eSpi specification as input. The newly generated eSpi document contains all the information that could be automatically inferred from the model and from the manually provided information.

It must be pointed out that refinement (step 2) can be repeated as many times as needed, until a satisfactory eSpi document is generated by the spi2Java refiner. However, in most cases, like in this example, one run of step 2 is enough.

3.3.2 Practice

Now we will show how the above presented steps can be accomplished in practice.

Step 1: Creating an eSpi Template.
The eSpi document for the *client* process can be created by using the `SpiTyperTextual` program. To obtain information for the *SpiTyperTextual* program options, it suffices to either call the program without parameters or by supplying the `-h` option, e.g.

```
java -jar SpiTyperTextual.jar -h
```

So, to create a first eSpi document from the symbol table of the PA Andrew process (i.e. step 1), the following command can be used:

```
java -jar SpiTyperTextual.jar
  -d ./data/
  -e ./data/espi_specification.xml
  -o ./testData/andrew/andrewPATemplate.xml
  -s ./testData/andrew/andrewPA.css
  -f
```

As a result, the eSpi document is created as "`./testData/andrew/andrewPATemplate.xml`" (since `-o` option specifies this as the output file), from the symbol table file (specified by the `-s` option). The `-f` option is supplied to instruct the program to silently overwrite the output file if this already exists. The `-d` option specifies the path to the SYSTEM DTD. This file is used by the tool to validate the eSpi document (which is an XML file). Finally, the `-e` option specifies the filename and path of the eSpi document.

Here is the content of the generated *andrewPATemplate* file.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE espi_types PUBLIC "espi_language.dtd" "./data/espi_language.dtd">
<espi_types>
<process id="0" name="pA_0">
```

```
<terms>
<term id="0" name="M_0" type="Message">
<codify>MessageSR</codify>
</term>
<term id="1" name="KeyStore_0" type="Channel">
<codify>This is an interface, please specify a subtype.</codify>
</term>
<term id="2" name="Na_1" type="Name">
<codify>NameSR</codify>
</term>
<term id="3" name="cAB_0" type="Channel">
<codify>This is an interface, please specify a subtype.</codify>
</term>
<term id="4" name="A_0" type="Message">
<codify>MessageSR</codify>
</term>
<term id="5" name="(A_0,Na_1)" type="Pair">
<codify>PairSR</codify>
</term>
<term id="6" name="xMSG_3" type="Shared Key Ciphered">
<codify>SharedKeyCipheredSR</codify>
<parameters>
<param name="algorithm" type="const">DES</param>
<param name="iv" type="const">12345678</param>
<param name="mode" type="const">CBC</param>
<param name="padding" type="const">PKCS5Padding</param>
<param name="provider" type="const">SunJCE</param>
</parameters>
</term>
<term id="7" name="B_0" type="Message">
<codify>MessageSR</codify>
</term>
<term id="8" name="kAB_5" type="Shared Key">
<codify>SharedKeySR</codify>
<parameters>
<param name="algorithm" type="const">DES</param>
<param name="strength" type="const">56</param>
<param name="provider" type="const">SunJCE</param>
</parameters>
</term>
<term id="9" name="_w0_6" type="Pair">
<codify>PairSR</codify>
</term>
<term id="10" name="xNa_7" type="Name">
<codify>NameSR</codify>
</term>
<term id="11" name="xk1AB_7" type="Shared Key">
<codify>SharedKeySR</codify>
<parameters>
<param name="algorithm" type="const">DES</param>
<param name="strength" type="const">56</param>
<param name="provider" type="const">SunJCE</param>
</parameters>
</term>
<term id="12" name="(xNa_7,xk1AB_7)" type="Pair">
<codify>PairSR</codify>
</term>
<term id="13" name="{Na_1}xk1AB_7" type="Shared Key Ciphered">
<codify>SharedKeyCipheredSR</codify>
<parameters>
<param name="algorithm" type="const">DES</param>
<param name="iv" type="const">12345678</param>
<param name="mode" type="const">CBC</param>
<param name="padding" type="const">PKCS5Padding</param>
<param name="provider" type="const">SunJCE</param>
</parameters>
</term>
<term id="14" name="dummy_10" type="Message">
<codify>MessageSR</codify>
</term>
<term id="15" name="(B_0,xk1AB_7)" type="Pair">
<codify>PairSR</codify>
</term>
<term id="16" name="{M_0}xk1AB_7" type="Shared Key Ciphered">
```

```
<codify>SharedKeyCipheredSR</codify>
<parameters>
<param name="algorithm" type="const">DES</param>
<param name="iv" type="const">12345678</param>
<param name="mode" type="const">CBC</param>
<param name="padding" type="const">PKCS5Padding</param>
<param name="provider" type="const">SunJCE</param>
</parameters>
</term>
</terms>
<expressions/>
</process>
</espi_types>
```

In the eSpi document, a `term` element contains additional information about a spi calculus term declared in the formal model. Three attributes are present in this element: `id` is a unique identifier for the term, used internally by the spi2Java tools; `name` is a human readable representation of the term, useful in order to uniquely identify it when manually modifying the eSpi document (a numeric suffix is added to the name that appears in the original spi calculus model); `type` contains the information about the type that has been statically assigned to the current term. The value of the `type` attribute must be present in the eSpi specification, and must be coherent with the use of the term in the model, as inferred by the spi2Java refiner. The `codify` element contains the name of a Java class implementing the encoding layer for the current term.

Step 2.1: Refining the eSpi document.
Once the `andrewPATemplate` has been created, there may be the need of specializing the type of some terms (step 2a). First of all, before modifying the file, it can be useful to create a sandbox copy of `andrewPATemplate.xml` that we call `andrewPAEdited.xml`.

In this specific case, the terms whose type needs to be refined, are:

- term 1 is a "Channel" that needs to be refined to "Java Key Store Channel".
- term 2 needs to be refined from "Name" to "Nonce".
- term 3 is a generic "Channel" (that is an interface). So, it needs to be refined. for example to "TCP/IP Channel".
- term 4 needs to be refined to "Identifier".
- term 7 needs to be refined to "Identifier".

Moreover, since the protocol establishes agreement on a key and we would like to allow the application to use the agreed key (after the protocol handshake), we need to specify that term 11 is a "protocol return parameter".

The content of the **andrewPAEdited** file, after the editing process, is reported here, followed by detailed comments about the modifications made.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE espi_types PUBLIC "espi_language.dtd" "./data/espi_language.dtd">
<espi_types>
<process id="0" name="pA_0">
<terms>
<term id="0" name="M_0" type="Message">
<codify>MessageSR</codify>
</term>
<term id="1" name="KeyStore_0" type="Java Key Store Channel">
<codify>JavaKeyStoreChannelSR</codify>
<parameters>
```

```
<param name="path" type="const">./andrewPA/</param>
<param name="keystore_filename" type="const">jce.keystore</param>
<param name="data_filename" type="const">jce.keystore.data</param>
<param name="type" type="const">JCEKS</param>
<param name="provider" type="const">SunJCE</param>
<param name="passwordmanager" type="const">it.polito.spi2java.spiWrapper.ConstantPassword</param>
</parameters>
</term>
<term id="2" name="Na_1" type="Nonce">
<codify>NonceSR</codify>
<parameters>
<param name="size" type="const">4</param>
</parameters>
</term>
<term id="3" name="cAB_0" type="Tcp/Ip Channel">
<codify>TcpIpChannelSR</codify>
<parameters>
<param name="host" type="const">localhost</param>
<param name="service" type="const">2006</param>
<param name="timeout" type="const">0</param>
</parameters>
</term>
<term id="4" name="A_0" type="Identifier">
<codify>IdentifierSR</codify>
</term>
<term id="5" name="(A_0,Na_1)" type="Pair">
<codify>PairSR</codify>
</term>
<term id="6" name="xMSG_3" type="Shared Key Ciphered">
<codify>SharedKeyCipheredSR</codify>
<parameters>
<param name="algorithm" type="const">DES</param>
<param name="iv" type="const">12345678</param>
<param name="mode" type="const">CBC</param>
<param name="padding" type="const">PKCS5Padding</param>
<param name="provider" type="const">SunJCE</param>
</parameters>
</term>
<term id="7" name="B_0" type="Identifier">
<codify>IdentifierSR</codify>
</term>
<term id="8" name="kAB_5" type="Shared Key">
<codify>SharedKeySR</codify>
<parameters>
<param name="algorithm" type="const">DES</param>
<param name="strength" type="const">56</param>
<param name="provider" type="const">SunJCE</param>
</parameters>
</term>
<term id="9" name="_w0_6" type="Pair">
<codify>PairSR</codify>
</term>
<term id="10" name="xNa_7" type="Name">
<codify>NameSR</codify>
</term>
<term id="11" name="xk1AB_7" type="Shared Key" return="true">
<codify>SharedKeySR</codify>
<parameters>
<param name="algorithm" type="const">DES</param>
<param name="strength" type="const">56</param>
<param name="provider" type="const">SunJCE</param>
</parameters>
</term>
<term id="12" name="(xNa_7,xk1AB_7)" type="Pair">
<codify>PairSR</codify>
</term>
<term id="13" name="{Na_1}xk1AB_7" type="Shared Key Ciphered">
<codify>SharedKeyCipheredSR</codify>
<parameters>
<param name="algorithm" type="const">DES</param>
<param name="iv" type="const">12345678</param>
<param name="mode" type="const">CBC</param>
<param name="padding" type="const">PKCS5Padding</param>
<param name="provider" type="const">SunJCE</param>
```

```
</parameters>
</term>
<term id="14" name="dummy_10" type="Message">
<codify>MessageSR</codify>
</term>
<term id="15" name="(B_0,xk1AB_7)" type="Pair">
<codify>PairSR</codify>
</term>
<term id="16" name="{M_0}xk1AB_7" type="Shared Key Ciphered">
<codify>SharedKeyCipheredSR</codify>
<parameters>
<param name="algorithm" type="const">DES</param>
<param name="iv" type="const">12345678</param>
<param name="mode" type="const">CBC</param>
<param name="padding" type="const">PKCS5Padding</param>
<param name="provider" type="const">SunJCE</param>
</parameters>
</term>
</terms>
<expressions>
</expressions>
</process>
</espi_types>
```

Term 2 (i.e. Na_1) has been automatically inferred by the spi2Java refiner to have the type *Name*. Indeed, no particular use is made of Na, such that it can be automatically inferred to belong to a more specialized type, and it can be correctly considered to be an opaque message; however, the informal protocol description specifies that Na is a nonce, so this information has to be manually specified in the eSpi document.

Since the Na term has been refined to the *Nonce* type, the default encoding layer has been modified accordingly (i.e. it has been changed from *NameSR* to *NonceSR*). Moreover, since the eSpi specification states that the *Nonce* type requires the "size" parameter, the `parameters` element has been added to include the set of parameters required by this type. Each `param` element has an attribute called `type`, which is used in order to specify whether the parameter is assigned at compile time (`type="const"`) or it must be resolved at run time (`type="var"`). If the parameter must be resolved at run time, then its value must be the identifier of another term that will contain the value of the parameter at run-time. In the reference example protocol, it is specified that the size of the nonce is statically fixed to 16 bytes, so "size" can be set as a `const` parameter in the eSpi document.

Term 1 (i.e. cAB_0) has been automatically inferred by the spi2Java refiner to have type *Channel*. However, this type corresponds to an interface and, hence, further refinement is needed. In particular, this channel has to be used as the communication channel between the two processes participating to the protocol. So, it needs to be refined as a channel that implements a network transport protocol. In this case, the chosen transport protocol is TCP/IP. So, the term is refined as *TCP/IP Channel*. Accordingly, the default encoding layer has been provided, i.e. *TcpIpChannelSR*. The same happens for the parameters of a *TCP/IP Channel* term type. In particular, the connection will be opened on 'localhost:2006', because of the `host` and `port` parameter values, and the connection will not have any timeout.

Term 3 is a *Channel* too, but it has a different aim: it represents the access point to a key store. So, the term is refined to *Java Key Store Channel* and, accordingly, *JavaKeyStoreChannelSR* is specified as the `codify` attribute. The path where the keystore files are stored has been set to 'user.dir', a keyword that indicates the current working directory; any other value that is not 'user.dir'

will be treated as a path. The Java Key Store implementation requires two files, one containing the proper keystore, and one containing accessory data: they are specified with the `keystore_filename` and `data_filename` parameters. Then, it has been chosen to use the Java Keystore implementation provided by the SunJCE provider. Finally, a password manager has been chosen: in this example it is used a password manager implementation that always returns an hard-coded password; in real applications the password manager will ask the user for a password.

Terms 4 and 7 have been automatically inferred by the spi2Java refiner to have type *Message*. Indeed, they are used by the protocol as actor identifiers of the entity running the processes. Therefore, they need to be refined as "Identifier". The default *IndentifierSR* type has been assigned as the `codify` type.

Term 11 has to be returned to the application after the protocol handshake. In fact, a security protocol can have some return parameters, such as the shared secrets generated during the protocol session, which shall be used after the end of the protocol session. In a session of the reference example protocol, the client agrees on a new key with the server. The key can be used later by the application to cypher messages. For this reason, the term $xk1AB_8$ is set as a protocol session return parameter by means of the `return="true"` attribute assignment.

In general, as many `return` parameters as needed can be declared in an eSpi document, and the spi2Java code generator will take care of making all protocol return parameters available to the Java code that is executed after a successful protocol session.

An important difference between the client and server roles concerns the communication channel: the server process must behave as a responder, while the client must behave as an initiator. Therefore, in the server case, the `server` attribute is set, to specify that the process owning this term will act as a responder on the cAB_0 channel (using the responder channel implementation provided in the *TcpIpServer* Java class), so the spi2Java code generator will take this into account. Since no term in the *Client* process has been manually given a `server` attribute, the client will be implemented as an initiator. Indeed, it is needed only in the server side of the protocol. The following excerpt of XML shows the eSpi definition of the server communication channel.

```
<term id="1" name="cAB\_0" type="Tcp/Ip Channel" server="TcpIpServer">
<codify>TcpIpChannelSR</codify>
<parameters>
<param name="host" type="const">localhost</param>
<param name="service" type="const">2006</param>
<param name="timeout" type="const">0</param>
</parameters>
</term>
```

Only when a channel is set as 'server', it is possible to specify the keyword 'any' for the `host` parameter, letting the server listen on any available network interface.

Remark: It may be argued that manual modification of the eSpi document is not very user friendly. This is true; however the whole spi2Java tool, which is currently a prototype, has been designed to be an integrated development environment (IDE). In this context, a comfortable user interface could accept user input, and then could transparently handle XML documents, automatically filling default values or adding required elements, as defined in the eSpi

specification. With this design in mind, the use of the machine readable XML document format, and the definition of the eSpi specification get even more importance. For example, when the user refines the type of Na from *Message* to *Nonce*, the IDE, according to the eSpi specification, can automatically change the default encoding layer, and can add all the required parameters for the new type, filling them with default values, or asking for custom values.

Step 2.2: Using the refined eSpi document.
Now that the `andrewPAEdited.xml` file has been modified, the *SpiTyperTextual* program has to be run again to check that the file does not contain any syntactic error and that there are no refinements that are not admissible by the Java class hierarchy. Moreover, the refiner could exploit the added information in order to further refine other terms. Note that other terms may also be refined by the typer as a consequence. The *SpiTyperTextual* can be invoked in this way:

```
java -jar SpiTyperTextual.jar
  -d ./data/
  -e ./data/espi_specification.xml
  -o ./testData/andrew/andrewPAFinal.xml
  -x ./testData/andrew/andrewPAEdited.xml
  -s ./testData/andrew/andrewPA.css
  -f
```

It is worth noting that, now, the andrewPAEdited.xml file is being provided as an input file to the tool (with the -x option), while a new output file is being specified (i.e. *andrewPAFinal.xml*). This file contains the refined specification. In this case it may be useful to perform a diff between the andrewPAEdited.xml and the andrewPAFinal.xml files to see if there are some differences, i.e. if some terms have been refined as a consequence of the refinement of other terms. In this case, only one term has been refined. In particular, term 10 (i.e. xNa_7) has been correctly refined to Nonce. So, since there is no need of refining any other term in the specification, it is possible to move further and proceed with code generation. Anyway, should the specification be not yet complete, step 2 could be repeated until a satisfying specification is created.

3.4 Implementing the Java Application

Once the final eSpi documents of the prototype version are done, the spi2Java code generator is used in order to obtain a Java implementation of the given spi calculus model, refined with the information contained in the coupled eSpi document.

The generated code uses classes and methods provided by the *spiWrapper* Java library (previously called *secureClasses* in [5]).

It is worth noting that the spi2Java code generator does not only generate the security critical Java code implementing the spi calculus model; instead, it also generates complete application templates that can be compiled and executed without any manual modification. This strongly reduces user interaction, enabling agile prototyping.

The spi2Java code generator can currently implement applications using two different architectures, whose flowcharts are reported in figure 4. If an application must act as an initiator, that is, no `term` element has the `server` attribute in the eSpi document, then the spi2Java code generator automatically uses the

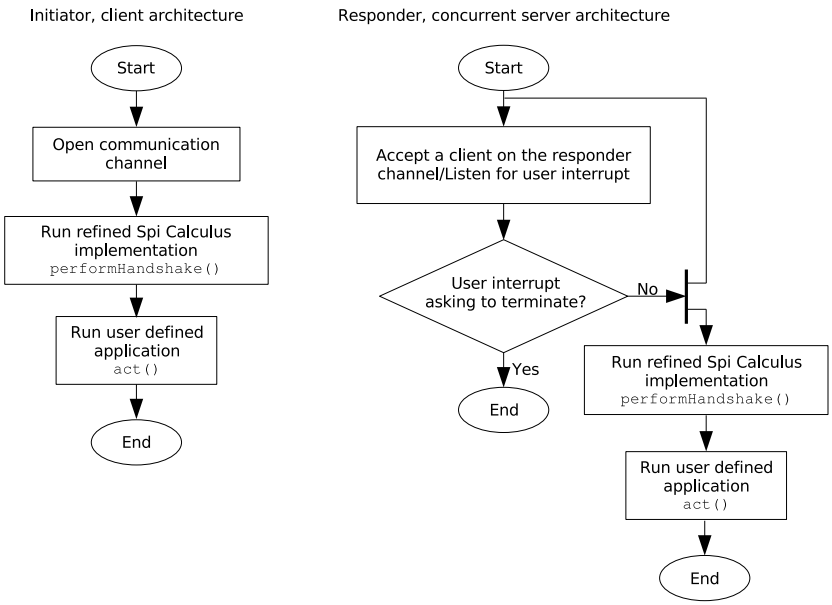

Figure 4: Flowchart of the initiator and responder architectures.

iterative client architecture. Otherwise the application must act as a responder on the channel having the `server` attribute, and the spi2Java code generator automatically uses the concurrent server architecture. The spi2Java code generator is designed so that it is easy to add new implementation architectures, such as concurrent crew servers (also known as "prefork") and the like.

It is worth noting that, in the application templates currently provided, while the `performHandshake()` method implements the logic of a protocol session, the `act()` method is executed only if the current protocol session ends successfully (that is, `performHandshake()` returns and does not throw an exception), and is initially empty. This method can be implemented by the user in order to perform any action that must be done after a successful end of the protocol session.

It must be also pointed out that, although the spi2Java code generator always generates code that can be compiled and executed without any modification, the input parameters of the `performHandshake()` method must be manually initialized before the program can be correctly executed, because no information can be automatically inferred on their contents. The method input parameters are all the free variables that are used in the process being implemented, with the exception of channels, that get configured automatically because the eSpi document already contains enough information for their initialization. For this reason, in the Andrew protocol example, variables M, A, B must be initialized in the *Client* process, while variable M must be initialized in the *Server* process. It is worth noting that input parameters can be initialized at compile time, or at run time, for example based upon user input.

3.4.1 Obtaining a protocol prototype implementation

When the eSpi specification file is in its final version, code generation can take place. The *JavaGeneratorTextual* program can be used to complete this task.

It is now shown how to get an early prototype of the protocol implementation, where the default encoding layer is used. Later on, we will show how a custom encoding layer can be implemented for both client and server. In order to generate the code that use the default encoding layer (i.e. Java Serialization), the *JavaGeneratorTextual* program has to be fed with the name of the package (i.e. it.polito.spi2java.spiWrapperSR) that contains the implementation of the encoding layer. This can be done by using the `-i` option, that specifies the name of packages that will be imported by the generated code. If many packages need to be imported, their names can be written in a text file, one per line, and the text file is passed to the *JavaGeneratorTextual* program by the `-j` option.

The *JavaGeneratorTextual* program can be invoked by using the following command:

```
java -jar JavaGeneratorTextual.jar
  -e ./data/espi_specification.xml
  -i it.polito.spi2java.spiWrapperSR
  -o ./testCode/andrew/andrewPAGeneratedJava/
  -p andrewPAGeneratedJava
  -s ./testData/andrew/andrewPA.css
  -t ./data/javaGenerator/
  -x ./testData/andrew/andrewPAFinal.xml
```

An explanation of the used options follows. The `-e` option specifies the path to the eSpi specification file. The `-i` option is used to import the *it.polito.spi2java.spiWrapperSR* package. The `-o` option specifies the path where the tool must place generated files. In this example, a nested directory is specified (i.e. andrewPAGeneratedJava). The `-p` option specifies the name of the package for the generated code. It is important to note that Java requires that the name of the package equals the name (i.e. andrewPAGeneratedJava) of the directory where the Java files are stored. The `-s` option specifies where the symbol table file is located. The `-t` option specifies where the tool can find some template files that it uses. The `-x` option provides the path to the final version of the eSpi document, for the process that has to be implemented.

The Java code generator produces the following files for a client implementation:

`pA_0_Main.java` : contains the main that initializes the parameters of the protocol and that invokes the protocol and then the application.

`pA_0_Protocol.java` : contains the implementation of a protocol session.

`pA_0_Application.java` : contains the skeleton of the application that is invoked after each protocol session execution.

Similarly, the *JavaGeneratorTextual* program can be invoked to generate the code for the *server* process, by using the following command:

```
java -jar JavaGeneratorTextual.jar
  -e ./data/espi_specification.xml
```

```
-i it.polito.spi2java.spiWrapperSR
-o ./testCode/andrew/andrewPBGeneratedJava/
-p andrewPBGeneratedJava
-s ./testData/andrew/andrewPB.css
-t ./data/javaGenerator/
-x ./testData/andrew/andrewPBFinal.xml
```

In this case, the generated files are:

`pB_0_Main.java` : contains the main that starts a server process, listening for clients.

`pB_0_Callback.java` : contains the code that is called once a client request has been accepted. This code initializes the parameters of the protocol and invokes the protocol and then the application.

`pB_0_Protocol.java` : contains the implementation of a protocol session.

`pB_0_Application.java` : contains the skeleton of an application that is invoked after each protocol session execution.

It is worth noting that the Java code generator does not know what are the values of the protocol parameters and, hence, their initialization must be manually provided. Therefore, if you try to run the generated code without providing parameters, the generated code execution stops, even before the code actually implementing the spi calculus model is run. So, in order to initialize the parameters, with respect to the *client* process of the Andrew protocol, the `pA_0_Main.java` file needs to be modified. In particular, it is necessary to change these statements

```
Message M_0 = null;
Identifier A_0 = null;
Identifier B_0 = null;
```

with, for example, the following statements

```
Message M_0 = new MessageSR("My message".getBytes());
Identifier A_0 = new IdentifierSR("A Role");
Identifier B_0 = new IdentifierSR("B Role");
```

It is worth noting that the new operators used in these statements refer to target classes that provide concrete data types, i.e. classes that include the implementation of an encoding layer (e.g. the *MessageSR* and the *IdentifierSR*). Classes belonging to the *it.polito.spiWrapper* package, such as *Message*, cannot be instantiated, because they are abstract classes. In this case, the classes using Java Serialization as the default encoding layer have been used (i.e. classes belonging to the *it.polito.spiWrapperSR* package). The names of these classes slightly differ from those included in the *it.polito.spiWrapper* package, because their name ends with the `SR` post-fix.

Let us now show how the protocol logic expressed in spi calculus translates into Java code. As an example, an excerpt of the code (part of the `performHandshake()` method body) emitted by the spi2Java code generator is reported below. This code implements the refined spi calculus model of the client in its prototype version, that uses the default encoding layer. It is worth noting that the spi2Java code generator has automatically added comments to

the code. So, they both improve code readability and make clear the mapping between each spi calculus statement and its Java implementation. In particular, it can be noted that each spi calculus statement is mapped on a few corresponding Java statements, and all the operations on a spi calculus term are handled by methods of the spiWrapper class implementing the type assigned to the term.

```
public List<Message> performHandshake(Message M_0, Identifier A_0,
    Identifier B_0, Channel KeyStore_0, Channel cAB_0)
    throws SpiWrapperException {
    /* Possibly thrown exception declaration. */
    SpiWrapperException _exceptionThrown = null;

    /* Declare and initialize the return parameters vector. */
    Vector<Message> _return = new Vector<Message>(1, 0);
    _return.setSize(1);

    /* Add all input parameters which must be returned to the _return vector */

    /* Declare new channels. */
    try {
        /* Here starts the real Spi protocol. */
        /* cAB_0<(A_0,Na_1)>. */
        Nonce Na_1 = new NonceSR("4");
        Pair PAIR_A_0_Na_1 = new PairSR(A_0, Na_1);
        cAB_0.send(PAIR_A_0_Na_1);

        /* cAB_0(xMSG_3). */
        SharedKeyCiphered xMSG_3 = (SharedKeyCiphered) cAB_0.receive(new SharedKeyCipheredSR());

        /* KeyStore_0<B_0>. */
        KeyStore_0.send(B_0);

        /* KeyStore_0(kAB_5). */
        SharedKey kAB_5 = (SharedKey) KeyStore_0.receive(new SharedKeySR());

        /* case xMSG_3 of {_w0_6}kAB_5 in */
        Pair _w0_6 = (Pair) xMSG_3.decrypt(new PairSR(), kAB_5, "DES",
                "12345678", "CBC", "PKCS5Padding", "SunJCE");

        /* let (xNa_7,xk1AB_7) = _w0_6 in */
        Nonce xNa_7 = (Nonce) _w0_6.getLeft();
        SharedKey xk1AB_7 = (SharedKey) _w0_6.getRight();
        _return.set(0, xk1AB_7);

        Pair PAIR_xNa_7_xk1AB_7 = new PairSR(xNa_7, xk1AB_7);

        /* [ xNa_7 is Na_1 ] */
        if (!xNa_7.equals(Na_1)) {
            throw new MatchException(
                "Match between \"xNa_7\" and \"Na_1\" failed.");
        }

        /* cAB_0<{Na_1}xk1AB_7>. */
        SharedKeyCiphered SH_ENC_Na_1_xk1AB_7 = new SharedKeyCipheredSR(Na_1,
                xk1AB_7, "DES", "12345678", "CBC", "PKCS5Padding", "SunJCE");
        cAB_0.send(SH_ENC_Na_1_xk1AB_7);

        /* cAB_0(dummy_10). */
        Message dummy_10 = (Message) cAB_0.receive(new MessageSR());

        /* KeyStore_0<(B_0,xk1AB_7)>. */
        Pair PAIR_B_0_xk1AB_7 = new PairSR(B_0, xk1AB_7);
        KeyStore_0.send(PAIR_B_0_xk1AB_7);

        /* cAB_0<{M_0}xk1AB_7>. */
        SharedKeyCiphered SH_ENC_M_0_xk1AB_7 = new SharedKeyCipheredSR(M_0,
                xk1AB_7, "DES", "12345678", "CBC", "PKCS5Padding", "SunJCE");
        cAB_0.send(SH_ENC_M_0_xk1AB_7);
```

By now, although the used encoding scheme may not be compliant with the protocol description, the prototype programs are fully implementing the proto-

col logic, so they can be used to test the protocol behavior and functionality. This test step is important, because a protocol could have been formally verified and resulted to be safe, even if it is not functional, and thus useless. For example, suppose a protocol where all sessions abort at the very beginning, because of a wrong design. Then this protocol is probably satisfying secrecy (and possibly authentication) requirements, because the intruder cannot get the secret, since no message is ever exchanged. However, such a protocol is useless.

Once protocol functionality has been tested on the prototype applications, the encoding layer can be implemented, in order to obtain fully functional and interoperable applications.

3.4.2 Customizing the Encoding Layer

In order to create the encoding layer, four abstract methods declared in the spiWrapper classes must be implemented by the programmer for each type of encoding that is required by the specification. More precisely, the four methods can be implemented by extending the spiWrapper class representing the type for which the encoding scheme is going to be written. It is worth noting that this approach isolates hand-written code with respect to automatically generated code.

The four methods that must be implemented are:

- `_encodePayload()`;
- `_serialize()`;
- `decodePayload()`;
- `deSerialize()`.

The first method is responsible for translating the internal representation of a term into the payload, encoded as requested by the informal protocol specification. The second method is used to add the necessary headers and trailers to the payload. This approach gives high flexibility by allowing different and independent encodings for cryptographic and networking operations.

The third and fourth methods are dual with respect to the first and second ones. `deSerialize()` extracts the payload from the whole data sent by the other parties, and `decodePayload()` transforms the encoded payload into the internal representation of the term.

4 Spi2Java Correlated Tools

This section enumerates and briefly describes some minor tools that are included in the spi2Java framework. For a reference of all options available in each tool (included the main tools explained above), please refer to the online help.

4.1 Key Store Generator

Sometimes, security protocols require that actors know in advance some protocol keys (by storing them in a keystore) before a protocol run starts, rather than filling the keystore during the protocol run. The 'andrew' protocol, used as

reference example, falls in the aforementioned case. For this reason, an "off-line" (that is, before a protocol run is started) tool that fills a keystore is provided. It can be used by issuing the following command:

```
java -cp .:../../spiWrapper.jar
     "keystoreGenerator.CreateJavaKeyStore"
```

4.2 Typer by Default

Some eSpi types, for instance *Channel* or *Hashing*, are abstract, and cannot be instantiated. Moreover, when automatically inferring types, it is not possible to choose an appropriate concrete type for these abstract types. When such abstract types are inferred, manual refinement becomes necessary in order to let the code compile and execute. However, if agile prototyping is aimed to, or all abstract types would be refined to the same concrete type, then it is possible to use the *TyperByDefault* program, that automatically refines each abstract type to a default concrete type.

The mapping between each abstract type and its default concrete type can be specified in a configuration file, one mapping per line; lines starting with `#` are considered as comments. The following is the content of an example mapping file:

```
# This is a comment line
# The format is "AbstractType:DefaultConcreteType"
Channel:Tcp/Ip Channel
Hashing:Cryptographic Hashing
```

An example of tool invocation follows.

```
java -jar TyperByDefaultTextual.jar
  -d data/
  -e data/espi_specification.xml
  -n data/defaultMapping
  -s testData/andrew/andrewPA.css
  -x testData/andrew/andrewPATemplate.xml
```

4.3 eSpi Merger

Sometimes it happens that, when manual refinement of the eSpi document is almost complete, one discovers some errors (e.g. a typo, or a wrong statement) in the protocol specification. Fixing the protocol leads to a new eSpi document, which must be manually refined from scratch again.

In order to solve this issue, the *eSpiMerger* program can be used. It accepts two file, a *new* file and an *old* file. The *new* file is the eSpi document obtained from the correct protocol specification, thus containing correct but unrefined terms. The *old* file is the eSpi document obtained from the erroneous protocol specification, thus containing wrong, but refined terms. The *eSpiMerger* program will try to refine the terms in the *new* file, by looking at how similar terms in the *old* file had been refined.

Unfortunately, the eSpi merger program is XML Schema based, so it is necessary to change the heading of the eSpi documents (telling them to use a

schema, and not a DTD) before passing them to this tool, and then bring the original heading back, before using other spi2java tools.

The header will likely change as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE espi_types PUBLIC "espi_language.dtd"
  "./data/espi_language.dtd">
<espi_types>

<?xml version="1.0" encoding="UTF-8"?>
<espi_types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="./data/espiXMLSchema-1.xsd">
```

An example of tool invocation follows.

```
java -jar espiMerger.jar
     -a newSpec.xml
     -b oldRefinedSpec.xml
     -e data/espi_specification.xml
     -o result.xml -f
```

4.4 Spi Source Cleaner

The spi2java tools accept an enriched version of the spi calculus, which allows comments, and free variables declaration. However, some tools like the S^3A formal verification tool, accept a plain spi calculus syntax, without comments or free variables declaration.

In order to "clean up" an enriched spi calculus code, the *SpiSourceCleaner* program can be used. For instance, the command

```
java -jar spiSourceCleaner.jar andrewPA.txt
```

will print on standard output the cleaned version of the protocol.

References

[1] L. Durante, R. Sisto, and A. Valenzano, "Automatic testing equivalence verification of spi calculus specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 12, no. 2, pp. 222–284, 2003.

[2] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: The spi calculus," in *ACM Conference on Computer and Communications Security*, 1997, pp. 36–47.

[3] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, parts I and II," *Information and Computation*, vol. 100, no. 1, pp. 1–77, 1992.

[4] A. Pironti and R. Sisto, "An experiment in interoperable cryptographic protocol implementation using automatic code generation," in *Computers and Communications*, 2007, pp. 839–844.

[5] D. Pozza, R. Sisto, and L. Durante, "Spi2java: Automatic cryptographic protocol java code generation from spi calculus," in *International Conference on Advanced Information Networking and Applications*, 2004, pp. 400–405.